**Dependency Injection in Android**

Dependency Injection (DI) in Android is a design pattern used to reduce tight coupling between classes and promote better modularity and testability. It involves providing dependencies (objects a class needs) from the outside rather than having the class instantiate them itself.

**🔧 Why Use Dependency Injection in Android?**

* **Easier to manage dependencies**
* **Improves testability (mocks can be injected)**
* **Encourages cleaner, modular code**
* **Reduces boilerplate code in larger projects**

**✅ Common DI Frameworks in Android**

1. **Hilt (Recommended by Google)**
2. **Dagger (used under the hood by Hilt)**
3. **Koin (lightweight, Kotlin-based)**
4. **Manual DI (simple projects or for learning purposes)**

**1. ✅ Hilt (Recommended by Google)**

Hilt is built on top of Dagger and simplifies DI setup in Android apps.

**🛠️ Setup (Gradle)**

gradle

CopyEdit

// Project-level build.gradle

buildscript {

dependencies {

classpath "com.google.dagger:hilt-android-gradle-plugin:2.50"

}

}

// App-level build.gradle

plugins {

id 'com.android.application'

id 'kotlin-kapt'

id 'dagger.hilt.android.plugin'

}

dependencies {

implementation "com.google.dagger:hilt-android:2.50"

kapt "com.google.dagger:hilt-compiler:2.50"

}

**✍️ Example**

kotlin

CopyEdit

// Dependency to be injected

class UserRepository @Inject constructor() {

fun getUser() = "John Doe"

}

// ViewModel using DI

@HiltViewModel

class MainViewModel @Inject constructor(

private val userRepository: UserRepository

) : ViewModel() {

fun getUserName() = userRepository.getUser()

}

kotlin

CopyEdit

// Application class

@HiltAndroidApp

class MyApplication : Application()

// Activity

@AndroidEntryPoint

class MainActivity : AppCompatActivity() {

private val viewModel: MainViewModel by viewModels()

}

**2. 🪶 Koin (Kotlin-Specific, Lightweight)**

gradle

CopyEdit

implementation "io.insert-koin:koin-android:3.5.0"

kotlin

CopyEdit

// Define your module

val appModule = module {

single { UserRepository() }

viewModel { MainViewModel(get()) }

}

// Start Koin in Application

class MyApp : Application() {

override fun onCreate() {

super.onCreate()

startKoin {

androidContext(this@MyApp)

modules(appModule)

}

}

}

**3. 🧱 Manual DI (No Framework)**

Good for small projects or learning.

kotlin

CopyEdit

class MainActivity : AppCompatActivity() {

private val userRepository = UserRepository()

private val viewModel = MainViewModel(userRepository)

}

**🧪 Benefits for Testing**

You can easily inject mocks during unit tests:

kotlin

CopyEdit

val fakeRepo = FakeUserRepository()

val viewModel = MainViewModel(fakeRepo)

**Summary**

| **Framework** | **Pros** | **Cons** |
| --- | --- | --- |
| **Hilt** | Official, powerful, scales well | Steeper learning curve |
| **Koin** | Simple, no code generation | Slightly slower at runtime |
| **Manual** | Simple, full control | Not scalable |

Let me know if you want a complete working sample project or integration tips!

“Used to reduce tight coupling between classes and promote better modularity and testability. It involves providing dependencies (objects a class needs) from the outside rather than having the class instantiate them itself.”

Let’s break down this concept in a simpler and clearer way:

**📌 What is Dependency Injection?**

Dependency Injection (DI) is a **design pattern** where one object **does not create** the objects it depends on (its **dependencies**) by itself. Instead, those dependencies are **given (injected)** to it from the outside.

**📦 Example (Without DI – Tightly Coupled Code)**

kotlin

CopyEdit

class Engine {

fun start() = "Engine started"

}

class Car {

private val engine = Engine() // ❌ Car creates its own dependency

fun drive() {

println(engine.start())

}

}

In this case, Car is **tightly coupled** to Engine because:

* It knows how to create an Engine.
* You can’t easily replace Engine with a mock or a different implementation.

**✅ Example (With DI – Loosely Coupled Code)**

kotlin

CopyEdit

class Engine {

fun start() = "Engine started"

}

class Car(private val engine: Engine) { // ✅ Engine is passed in

fun drive() {

println(engine.start())

}

}

Now:

* Car doesn’t care **where the Engine comes from**.
* You can **inject a mock Engine** for testing.
* This makes the code **easier to maintain, extend, and test**.

**🔧 Why Is This Useful?**

**✅ Modularity:**

Each class does **only one job** and doesn't manage other classes’ lifecycles.

**✅ Testability:**

You can easily replace dependencies with **mock or fake implementations** during testing.

**✅ Maintainability:**

You can update or swap dependencies **without changing the class that uses them**.

**🧠 Real-World Analogy**

Imagine a chef in a restaurant:

* ❌ **Without DI**: The chef also grows vegetables, raises animals, buys spices... too much responsibility.
* ✅ **With DI**: Someone delivers ingredients to the chef — now the chef only focuses on cooking.

**🔁 In Android**

Instead of each Activity or ViewModel creating things like Repository or Database objects directly, **DI frameworks like Hilt or Koin** provide those objects **automatically**.

Visual diagram or a code walkthrough with DI in an Android app (like ViewModel + Repository + Retrofit)

— let's go deeper with a **simple Android-style DI example** using ViewModel, Repository, and a fake data source to show **how DI helps**.

**🚗 Scenario: You have a ViewModel that shows user data.**

**📦 Without DI (Bad: Tightly Coupled)**

kotlin

CopyEdit

class UserRepository {

fun getUser(): String = "John Doe"

}

class UserViewModel : ViewModel() {

private val repository = UserRepository() // ❌ tightly coupled

fun getUserName(): String {

return repository.getUser()

}

}

* UserViewModel creates its own UserRepository.
* You can't easily test UserViewModel with a fake repository.
* Changing the repository means editing the ViewModel.

**✅ With DI (Good: Loosely Coupled)**

kotlin

CopyEdit

class UserRepository {

fun getUser(): String = "John Doe"

}

class UserViewModel(private val repository: UserRepository) : ViewModel() {

fun getUserName(): String {

return repository.getUser()

}

}

Now, you can create it like this:

kotlin

CopyEdit

val realRepo = UserRepository()

val viewModel = UserViewModel(realRepo)

And in tests:

kotlin

CopyEdit

class FakeUserRepository : UserRepository() {

override fun getUser(): String = "Fake User"

}

val fakeRepo = FakeUserRepository()

val testViewModel = UserViewModel(fakeRepo)

🎉 **No need to change the ViewModel class** — you just pass in whatever repo you want!

**🤖 In Android with Hilt**

Let’s say we use Hilt to inject everything automatically:

**1. Define your repository**

kotlin

CopyEdit

class UserRepository @Inject constructor() {

fun getUser(): String = "John Doe"

}

**2. Inject it into your ViewModel**

kotlin

CopyEdit

@HiltViewModel

class UserViewModel @Inject constructor(

private val repository: UserRepository

) : ViewModel() {

fun getUserName() = repository.getUser()

}

**3. Use it in Activity or Fragment**

kotlin

CopyEdit

@AndroidEntryPoint

class MainActivity : AppCompatActivity() {

private val viewModel: UserViewModel by viewModels()

}

**4. Application class**

kotlin

CopyEdit

@HiltAndroidApp

class MyApplication : Application()

**📌 Summary**

| **Without DI** | **With DI** |
| --- | --- |
| Classes create their own dependencies | Dependencies are passed from outside |
| Harder to test and change | Easy to test, reuse, and change |
| Tightly coupled | Loosely coupled |

Let me know if you want a **complete sample Android project**, or if you're using **Kotlin, Java, or Jetpack Compose** — I can tailor it to that.

**1️⃣ What does "tightly coupled" mean?**

Imagine two classes: Car and Engine.

kotlin

CopyEdit

class Engine {

fun start() = "Engine started"

}

class Car {

private val engine = Engine()

}

Here, Car creates its own Engine. So:

* Car is **tightly coupled** to Engine.
* If you want to change how Engine works, or replace it with a fake one (like in tests), you have to **change the Car class itself**.

💬 **Tightly coupled** means:

Two pieces of code are stuck together. If you change one, the other breaks.

**2️⃣ Why is it called a "dependency"?**

A **dependency** is just:

Something your class *needs* in order to do its job.

Examples:

* A ViewModel depends on a Repository
* A Car depends on an Engine
* A LoginScreen depends on a LoginService

Think of it like this:

"I can't work unless you give me this thing I depend on."

So the **Repository** is a *dependency* of the ViewModel.

**3️⃣ What does "Dependency Injection" actually mean?**

**Injection** means: You *give* the dependency to the class **from outside** — you *don’t* let the class create it.

Like this:

kotlin

CopyEdit

class Car(private val engine: Engine) {

fun drive() = engine.start()

}

Now, someone else (outside the Car) decides which Engine to give it.

**4️⃣ "Testing with fake things" – what does that mean?**

Let’s say your real app uses a real Repository that talks to the internet.

In tests, you **don’t want** to actually make network calls — they’re slow and can fail randomly.

Instead, you create a **fake** or **mock** repository:

kotlin

CopyEdit

class FakeUserRepository : UserRepository() {

override fun getUser(): String = "Fake User"

}

Then test your ViewModel like this:

kotlin

CopyEdit

val viewModel = UserViewModel(FakeUserRepository())

assert(viewModel.getUserName() == "Fake User")

✅ No network  
✅ Fast test  
✅ No real server needed

This is only possible because **you injected the repository** — you didn’t hardcode it inside UserViewModel.

**🔄 Summary (Simple Definitions)**

| **Term** | **What it Means** |
| --- | --- |
| **Dependency** | Something a class needs (like a Repository or Engine) |
| **Injection** | Giving that thing to the class from the outside |
| **Tightly Coupled** | One class creates and controls another — hard to change/test |
| **Loosely Coupled** | Classes just use what they’re given — easy to test and swap |
| **Fake** | A pretend version used in tests instead of the real thing |

Dependency Injection (DI) is a **popular solution**, but it's not the *only* way to manage dependencies. Here are other approaches — with pros, cons, and when you might use them.

**✅ 1. Service Locator Pattern**

A central object (like a ServiceLocator class) holds and gives out dependencies.

**Example:**

kotlin

CopyEdit

object ServiceLocator {

val userRepository = UserRepository()

}

class UserViewModel {

private val repo = ServiceLocator.userRepository

}

**✅ Pros:**

* Simple to implement
* Central control of dependencies

**❌ Cons:**

* Still tightly coupled
* Hidden dependencies (class looks independent but secretly fetches dependencies)
* Harder to test than DI

**✅ 2. Manual Instantiation (Factory Pattern)**

You manually create and pass in dependencies — no framework.

**Example:**

kotlin

CopyEdit

class AppFactory {

fun createUserViewModel(): UserViewModel {

val repo = UserRepository()

return UserViewModel(repo)

}

}

**✅ Pros:**

* Full control
* Easy to understand
* Good for small projects

**❌ Cons:**

* Gets messy as app grows
* Lots of boilerplate
* Hard to manage dependencies with many classes

**✅ 3. Singleton Pattern**

One instance is shared across the app.

**Example:**

kotlin

CopyEdit

object UserRepository {

fun getUser() = "John Doe"

}

class UserViewModel {

fun getUserName() = UserRepository.getUser()

}

**✅ Pros:**

* Simple
* Shared state
* Easy to access

**❌ Cons:**

* Hidden coupling
* Harder to test or replace
* Global state can lead to bugs

**✅ 4. Service Container / IoC Container**

Similar to DI frameworks — the app asks a container for what it needs.

This is what frameworks like **Spring** (in Java) or **Koin** (in Kotlin) use behind the scenes.

**✅ 5. Hardcoded Dependencies (No Pattern)**

This is the default — just create what you need in every class:

kotlin

CopyEdit

class ViewModel {

private val repo = UserRepository()

}

**❌ Problems:**

* No flexibility
* Not testable
* Changes require rewriting multiple places

**🧠 Summary Table**

| **Method** | **Testable** | **Easy** | **Scales Well** | **Best For** |
| --- | --- | --- | --- | --- |
| **Dependency Injection (DI)** | ✅ | ⚠️ Medium | ✅✅ | Large, complex apps |
| **Service Locator** | ⚠️ Semi | ✅ | ❌ | Medium apps |
| **Manual (Factory)** | ✅ | ✅ | ❌ | Small apps |
| **Singletons** | ❌ | ✅ | ❌ | Global settings, constants |
| **Hardcoded** | ❌ | ✅ | ❌ | Learning/demo apps only |

Let’s walk through a **step-by-step transformation** from a tightly-coupled, hardcoded setup to clean, testable Dependency Injection — using a simple UserViewModel and UserRepository setup.

**🎯 Goal: Show user's name using a ViewModel and a Repository**

**🧱 Step 1: Hardcoded (Bad / Tightly Coupled)**

kotlin

CopyEdit

class UserRepository {

fun getUser(): String = "John Doe"

}

class UserViewModel {

private val repo = UserRepository() // ❌ creates its own dependency

fun getUserName(): String {

return repo.getUser()

}

}

**🚫 Problems:**

* Can’t swap UserRepository in tests
* Changing the repository forces you to edit UserViewModel

**🔧 Step 2: Constructor Injection (Manual DI)**

kotlin

CopyEdit

class UserRepository {

fun getUser(): String = "John Doe"

}

class UserViewModel(private val repo: UserRepository) {

fun getUserName(): String = repo.getUser()

}

kotlin

CopyEdit

// App setup

val repo = UserRepository()

val viewModel = UserViewModel(repo)

**✅ Benefits:**

* UserViewModel doesn’t care where the repo came from
* Easy to test:

kotlin

CopyEdit

class FakeUserRepository : UserRepository() {

override fun getUser(): String = "Fake User"

}

val testVM = UserViewModel(FakeUserRepository())

println(testVM.getUserName()) // Prints "Fake User"

**⚙️ Step 3: Framework-Based DI (Hilt)**

Use **Hilt** to inject everything automatically.

**1. Add Dependencies in build.gradle:**

gradle

CopyEdit

plugins {

id 'kotlin-kapt'

id 'dagger.hilt.android.plugin'

}

dependencies {

implementation "com.google.dagger:hilt-android:2.50"

kapt "com.google.dagger:hilt-compiler:2.50"

}

**2. Create UserRepository:**

kotlin

CopyEdit

class UserRepository @Inject constructor() {

fun getUser() = "John Doe"

}

**3. Create UserViewModel:**

kotlin

CopyEdit

@HiltViewModel

class UserViewModel @Inject constructor(

private val repo: UserRepository

) : ViewModel() {

fun getUserName() = repo.getUser()

}

**4. Enable Hilt in Application class:**

kotlin

CopyEdit

@HiltAndroidApp

class MyApp : Application()

**5. Inject into Activity or Fragment:**

kotlin

CopyEdit

@AndroidEntryPoint

class MainActivity : AppCompatActivity() {

private val viewModel: UserViewModel by viewModels()

}

**✅ Now:**

* No manual wiring
* Easy to test with fake objects
* Framework handles injection for you

**🚀 Recap**

| **Step** | **DI Style** | **Testable?** | **Scalable?** |
| --- | --- | --- | --- |
| Step 1 | Hardcoded | ❌ | ❌ |
| Step 2 | Manual Constructor DI | ✅ | ⚠️ |
| Step 3 | Hilt (Framework DI) | ✅✅ | ✅✅ |

DI in **Jetpack Compose** and **Kotlin Multiplatform (KMP)** works a bit differently due to how they handle UI and shared code, but the core principles remain the same: **inject dependencies instead of creating them directly**.

**🧩 Part 1: DI in Jetpack Compose**

Jetpack Compose works with any DI framework (like Hilt or Koin), but you handle injection slightly differently because Compose functions don't have constructors.

**✅ Hilt in Compose**

To inject dependencies into ViewModels inside Compose:

**1. Annotate your ViewModel:**

kotlin

CopyEdit

@HiltViewModel

class MainViewModel @Inject constructor(

private val repo: UserRepository

) : ViewModel()

**2. Use hiltViewModel() in Composable:**

kotlin

CopyEdit

@Composable

fun MyScreen() {

val viewModel: MainViewModel = hiltViewModel()

val name = viewModel.getUserName()

Text(text = name)

}

**3. Don't forget:**

* Annotate your Application with @HiltAndroidApp
* Use @AndroidEntryPoint on your Activity

**✅ Koin in Compose**

Koin is even more Compose-friendly — you just call getViewModel() inside your composable.

kotlin

CopyEdit

@Composable

fun MyScreen() {

val viewModel: MainViewModel = getViewModel()

val name = viewModel.getUserName()

Text(name)

}

**📦 Part 2: DI in Kotlin Multiplatform (KMP)**

KMP apps usually have a **shared module** (for business logic) and platform-specific code (for Android/iOS UI).

Since you can't use Hilt (Java-based) in shared code, most KMP apps use:

**✅ KMP-Friendly DI Frameworks:**

| **Framework** | **KMP Support** | **Notes** |
| --- | --- | --- |
| **Koin** | ✅ Full | Kotlin-first, great for KMP |
| **Kodein-DI** | ✅ | Lightweight, less popular recently |
| **Manual DI** | ✅ | Common in simple shared codebases |

**Example: DI in KMP with Koin**

**Shared Module:**

kotlin

CopyEdit

class UserRepository {

fun getUser(): String = "John Doe"

}

class MainViewModel(private val repo: UserRepository) {

fun getUserName(): String = repo.getUser()

}

**Koin Module (shared):**

kotlin

CopyEdit

val sharedModule = module {

single { UserRepository() }

factory { MainViewModel(get()) }

}

**Initialize Koin (Android side):**

kotlin

CopyEdit

startKoin {

androidContext(this@MyApp)

modules(sharedModule)

}

Then, inject on Android like normal:

kotlin

CopyEdit

val viewModel: MainViewModel = get()

On iOS, you write a bridge to get MainViewModel from shared module using the same Koin instance.

**🧠 Summary**

| **Context** | **DI Framework** | **Notes** |
| --- | --- | --- |
| **Jetpack Compose** | Hilt, Koin | Use hiltViewModel() or getViewModel() |
| **Kotlin Multiplatform** | Koin, Manual | Use Koin modules in shared code, no Hilt |